Abstract

Having humanoid robots retrieve objects on demand in the real world has long been an aspiration of robotics. Given access to such a robot with Imperial College London’s Robot DE NIRO, our goal was to develop a stable fetch routine with natural human-to-robot interaction. We built or integrated components for Audio I/O, Mapping and Navigation, Object Recognition, Grasping, and Face Recognition. We called upon these with a state machine and managed inter-component communication with Robot Operating System as middleware. We achieved our goals, and the resulting work offers a good base for further research on large, human-scale robots like Robot DE NIRO.

Word Count of 12,930 = ~26 pages, excluding headers and figures.

Link to full interactive documentation:
## Contents

1 Introduction
   1.1 Why Fetch? ................................................. 1
   1.2 The Robot ............................................. 1

2 Specification
   2.1 Original Requirements .................................... 3
   2.2 Expanded Subtasks .................................... 3

3 Design and Approach
   3.1 Components ........................................... 4
   3.2 Object-Oriented Design ................................ 4
   3.3 Chasing the MVP ...................................... 5
   3.4 Graceful Downgrades ................................... 6
   3.5 State Machine ......................................... 7
      3.5.1 Overview ........................................ 7
      3.5.2 Design Choices ................................ 7

4 Software Engineering ........................................... 9
   4.1 Project Management ..................................... 9
      4.1.1 Scrum ........................................... 9
      4.1.2 The Challenges of the First Half ................ 9
      4.1.3 The Progress of the Second Half ............... 10
      4.1.4 Project Management Tools ....................... 11
   4.2 Software Development ................................ 12
      4.2.1 Git ........................................... 12
      4.2.2 Testing ....................................... 13
      4.2.3 Documentation ................................ 16

5 Robot Operating System ........................................ 17
   5.1 ROS Overview ......................................... 17
   5.2 File Structure ....................................... 17
   5.3 The ROS Master ....................................... 17
   5.4 Asynchronous Communication: Publisher/Subscriber Model ........................................ 18
   5.5 Synchronous Communication: ROS Service Model ........................................ 18
   5.6 Experimenting with ROS ................................ 19

6 Component Features ........................................... 20
   6.1 Command Input (Owner: Clara Pouletty) ............... 20
   6.2 Audio Output (Owner: Nico Smuts) ..................... 21
   6.3 Mapping and Navigation (Owner: Nico Smuts) ....... 22
   6.4 Object Recognition (Owner: Kim Rants) ............... 24
   6.5 Grasping (Owner: John Lingi) ......................... 26
   6.6 Face Recognition (Owner: Fabian Falck) ............... 28
   6.7 Uncertainties ......................................... 30

7 Final Product ................................................. 32
   7.1 Areas for Further Research ......................... 32
   7.2 Project Fezzik ....................................... 32

8 Acknowledgements ............................................ 34

A Appendix .................................................... 37
1 Introduction

1.1 Why Fetch?

Project Fezzik sought to accomplish a classic goal of robotics: have a robot autonomously fetch an object. The paradigmatic use is for a home assistant robot. Consider someone less able-bodied, with mobility challenges. This human asks the robot to fetch, say, a bottle of medicine. The robot comprehends, navigates to the correct location, recognises the item, retrieves it, returns to the requester, and hands over the object. A simple, resilient, and thorough implementation of this fetch routine could help improve lives.

Fetching an object, effortless for most healthy people, actually consists of complex subtasks using sensors and actuators. In humans, our sensors are eyes, ears, and hands, while robots need cameras, wheels, and motors. Both of us use our own subcomponents for perceiving our environment and acting upon our environment.

This, then, was the software challenge posed by this project. As you will see, empowered with well-crafted hardware, this project was a success. This is how Project Fezzik runs: First, a human can initiate the task by asking our robot to fetch a medicine bottle. The robot will comprehend, remember the requester, and navigate to a pre-chosen location. Upon arrival, our robot may see a number of objects. As long as the bottle has the right marker on its label, the robot will pick it up. From there, the robot will return back to where it started, surrendering the object as long as the original requester is present.

Our implementation is limited by some of the noisiness of the real world. But it works – and it works today. In this report, we describe how we arrived at this point. We explain our design, our process, our tools, our techniques, our failures, and our adaptations. We have carefully documented and designed our project to ease the way for any future developers in our lab and beyond. We hope this project may one day lead to the domestic robot assistant that so many of us picture when we consider the future of robotics.

1.2 The Robot

We have worked on this project with Imperial College London’s Robot Intelligence Lab, led by Dr. Petar Kormushev. Our main tool in this project was the Lab’s largest and original robot, built by Dr. Kormushev: Robot DE NIRO.

Robot DE NIRO (Design Engineering’s Natural Interaction RObot) is constructed around the industrial Baxter robot arms built by Rethink Robotics [1]. These are a pair of linked, powerful arms with exchangeable grippers that boast a variety of native sensors: “eye-in-hand” cameras, infrared range sensors, accelerometers, interlocking motors, and actuator torque output measurement devices. The Lab augmented these arms with additional capabilities, including:

- a QUICKIE movable electric wheelchair base with an integrated Mbed microcontroller,
- a Microsoft Kinect depth camera,
- stereovision cameras with inbuilt microphones,
- a 2D Hokuyo LIDAR scanner,
- a Logitech multi-camera rig enabling 360-degree vision, and
- an Amazon Echo Dot for audio input.
All of these actuators and sensors together make up Robot DE NIRO. This variety gives DE NIRO a wide range of abilities, but that same variety entails complex integration. The data produced by any given sensor is only useful when combined with the others in the right order and in real-time processing. This integration is, fundamentally, a software challenge, and it is at the heart of our project.

Figure 1: Front view of Robot DE NIRO, as modified by the Lab.
2 Specification

2.1 Original Requirements

In our initial specification, we worked with Dr. Kormushev and our Department of Computing advisor, Professor Andrew Davison, to agree on goals. Originally, we chose to have DE NIRO accomplish four tasks:

- understand a request for an object,
- identify that object in the environment,
- retrieve the requested object, and
- return it to the requester.

Those are the highest level requirements, and those continue to be in place today. Robot DE NIRO cannot play fetch without passing through each of these steps, so each was a hard requirement for the success of our project.

2.2 Expanded Subtasks

During implementation, we adjusted and augmented how we accomplished each step. We extracted out implied steps into their own, capsuled actions. In this way, we split the four general tasks from our specification into a more measurable set of nine:

1. Initialise Robot
2. Idle
3. Listen for Instructions
4. Remember User
5. Navigate to Object Warehouse
6. Detect and Grasp Object
7. Return to Original Location
8. Seek Original User
9. Offer Object to User

These more specific steps became states we designed into our state machine. DE NIRO shifted from one state to the next on its way through the fetch routine. Each step suggests a multitude of implementation possibilities, each one with its own trade-offs.

Splitting these tasks helped us realise that there was a variety of ways to achieve them. There is far from only one way to navigate or to grasp, or to receive a command. In fact, some tasks, like navigation, may require repeat invocation with slightly different inputs, even as the overall state path moves sequentially through all nine. These states, in other words, need not map one-to-one with the actual features we built. This idea heavily influenced our design, which we explain in the next section.
3 Design and Approach

3.1 Components

We chose to identify distinct components that did not map perfectly to our states. It appeared that there would usually be one major action or idea per state, with some repeats. After considering the states, the components we build increasingly seemed to be the building blocks of our project. Their moderately independent natures means that although we chose to arrange them in this particular order for a fetch routine, another project may mix and match them in a different collection of states. The core components, though, should be functional and smooth however they would be used. These software features – distinct from the states listed in Section 2.2 – are:

1. Command Input
2. Audio Output
3. Mapping and Navigation
4. Object Recognition
5. Grasping
6. Face Recognition

This, then, is what we needed to build. Each of these pieces needed to work reliably and flexibly to fit together into our flow. This list was daunting. It forced us to comprehend that robotics is, by necessity, a fundamentally multidisciplinary activity. Each component is almost a project in itself. Indeed, we discuss them in more detail in Section 6. They all require different skills, use different software packages, and relate to different physical parts of the robot. We needed to develop proficiency with a disparate range of computing topics, including computer vision, natural language processing, control flow, networking, concurrency, and finite-state machines.

Furthermore, because most components were unique, there was little crossover benefit. Working on navigation turned out to involve 3D coordinates, transforms, and delicate devices that dispatched electric current to wheels. Face recognition, on the other hand, required manipulating real-time image streams, testing facial feature packages, and linking separate libraries' transform functions. Progress in the former did not make the latter any easier.

3.2 Object-Oriented Design

From the beginning, we hoped to make a product useful to future researchers. We structured our code to follow object-oriented principles by capsuling related functions and variables into methods and attributes. Our goal was to make our code more sustainable without becoming bloated.

The key idea to model was how components may communicate. To represent communications over ROS (see Section 5), we kept four primary parent classes, paired up to match to their mode of communication:

- ROSPublisher, ROSSubscriber
- ROSServiceServer, ROSServiceClient.

If a component used a particular communication modality, we created two component classes, each one inheriting directly from one in a linked pair of parents. The parents classes provide abstract functions which much be overwritten by child classes in order to communicate with any other classes. Furthermore, each class performs extra processing unique to itself in a unique collection of bespoke methods.
Figure 2 shows an extract of our UML class diagram (the full diagram is in Appendix A). To keep our class diagrams manageable, we eschewed ancillary classes, standard class members, and ROS utility constructs. We aimed to have an object-oriented structure that shows how our diverse components are united by the way they communicate.

3.3 Chasing the MVP

With so many components, we had to carefully consider how we would organise them. We needed a solution that would be simple, easy to experiment with, and well-suited to robotics. To link our components and our states, we therefore decided on two frameworks that permeated our project: first, the state machine (discussed in Section 3.5), and second, Robot Operating System [2], or ROS (explored in Section 5). The state machine gave us much-needed flexibility with our fetch routine. ROS allowed us to abstract away the concurrency and networking challenges with linking and running multiple shared computing devices. Our challenge here and throughout the project was to build useful components while keeping the integration burden light.

This, then, was where things stood early in our project. We knew what components we needed, and we knew the order in which to execute them. The question was whether and how we could build them all. It seemed natural at that stage to divide the components across our team of six engineers, with each person responsible for experimenting and studying his or her component. This we did, not anticipating just how many obstacles we would encounter. Both integrating existing packages and creating them from scratch proved difficult, even discouraging. We discuss each component, its challenges, and its alternatives in Section 6.

However, from a project design perspective, beginning by investigating the components was a fateful decision. It was not wrong – indeed, it still seems a sensible way to divide up a multi-pronged software project – but it did influence the rhythm of our work. We gained a deeper understanding of each
component, but delayed our integration across components. The extra subject area expertise yielded some benefits. For one, we could be more thorough. We typically tested multiple alternatives for a given feature, rather than just one. Our object recognition team, for example, tested five methods, weighing robustness, ease of integration, and simplicity. For another, we could more quickly uncover problem areas. We learned very early on, for example, that autonomous navigation was unstable and would be a major challenge.

Our design choice (and our hope) was to quickly prototype components and then stitch together an early minimum viable product. This is where we made a mistake. Getting working components was very challenging and slow-moving. With haphazard schedules, coursework filling our free time, and exams on the horizon, we stuck with the inertia of this original plan. We wish now that having noticed slow progress with the components, we had shifted our energies to building the more integrative features. Without pursuing full integration first, some working components became more complicated than necessary and others just seemed impossible with the current approach. Although we eventually righted ourselves, this led to some lost time and wasted effort.

Still, we attacked early challenges with self-organisation techniques, some of which were unique to robotics work. We describe our software engineering approaches in Section 4.

3.4 Graceful Downgrades

Even before we had spent months trying to build each component, we knew that some stood out as riskier or more challenging than others. From an early stage, we had to consider whether we could simplify some to still complete the full fetch process. Our conclusion was that it could still work – even without some steps or components – as long as we had satisfactory fallback options.

In other words, we needed to give each step a graceful downgrade. What if, for instance, DE NIRO could not identify the original requester in the SeekingUser state? One sensible downgrade might be for DE NIRO to proffer its object to the first human it encounters. That would be an acceptable loss. Similarly, we considered downgrades for each state, trying to soften any failure.

This philosophy of considering failure modes first proved useful: our final implementation incorporates multiple ideas that began as downgrades. As we learned time and time again, the world is noisy, and DE NIRO can get confused outside of a protected environment. Having a wide range of possibilities for how DE NIRO can seek to meet our expectations is actually more practical. In some cases, we could even automate the shift to downgrade alternatives, or simply offer both at the same time.
3.5 State Machine

3.5.1 Overview

The state machine is the governor of Project Fezzik. It manages the control flow of the entire fetch routine by invoking a component feature based on the current state. As such, it sits distinct and apart from the other components.

A finite-state machine is a well-known computational model. We chose to apply it in our project to easily manage our component invocations and to keep our design as simple as possible. By only considering the components and their respective classes necessary per state, we could reduce the number of topics to manage at any given moment and therefore the computational cost to run our program. Our state machine was able to stay relatively uncluttered, with a mostly linear progression through each state. ROS boasts the SMACH package, meant specifically to guide a robot through a state machine.

![Diagram of flow between the fetch process states](image)

Figure 3: Diagram of flow between the fetch process states

Without a state machine, our code may have been brittle and convoluted. Repeat-use components may have demanded a looping structure, or perhaps a long list of repeated calls. Without knowing which component was next in line, the program may have needed a series of computationally expensive polling I/Os, full of spinning while loops. Moreover, the communication from component to component – albeit still done through ROS – would have been more haphazard, perhaps flooded with global variables.

With a state machine, the robot can ignore extraneous inputs and stay focused on the task at hand. While navigating, all that matters is what the LIDAR sees. While listening, only audio input. Viewing the target object happens separately from moving the arms to that location. All this eases troubleshooting and simplifies control logic for the user.

3.5.2 Design Choices

Linear The state machine is set up to transition linearly between the states. This makes flow simple and deterministic. We originally implemented a more complex state machine, allowing it to switch among states based on the current state’s success, failure, or pre-emption by another. However, we found that much of this work was redundant. In the event of a successful or unsuccessful grasp, for example, DE NIRO would still progress on to returning to the origin. There were some failure points that did affect subsequent behaviour. A failure to find the original requester should not necessarily result in an offer. For convenience, we therefore decided to store state outcomes in the Parameter Server (explained in Section 5), allowing the brain to get those parameters and act accordingly.
Modular All states are built with the same structure. The state machine interacts with outside components via APIs. This offers a lot of flexibility, as states can be added, re-ordered, or removed without impacting the rest of the flow. For example, when the wheelchair base runs out of battery, we can simply redirect the state machine to simply skip over the portions requiring navigation. As we became more settled on which states to include, this same modularity was of great use.

Idle State Initially, the state machine was set up to rest in the supposedly quiet state of Listening. However, this meant the robot was always active and would trigger the moment it was given an instruction. With the highly sensitive object recognition or with audio input, accidental launches were very common. We felt a more stable design choice would be to introduce an Idling state that forces the user to deliberately engage the robot to begin the process. In a real-world deployment, constant listening would be a choice users could decide on.

Looping The final state, OfferingObject, is set to transition back to Listening so that users can make repeated requests. Without this loop, the robot would have to be shut down and restarted after each run. This is impractical, especially considering the use case for the mobility-challenged.

Minimal The state machine focuses on control flow and almost nothing else. Although SMACH supports state to state data transfer, we decided to keep our state machine slim. No data is passed exclusively through it. Rather, we make use of the optimised conduits in ROS. This goes hand-in-hand with our modular design and allows rapid prototyping without the need for extraneous local variables to indicate state. In addition, there are no busy waits, loops, or other control flow-determining statements here. The job of the state machine is to take in the current state and muster the appropriate components.
4 Software Engineering

4.1 Project Management

4.1.1 Scrum

As a practicum for agile software development, Project Fezzik served us well. With such a wide array of challenges before us, we tested, applied, and modified a variety of software engineering practices to best match our needs. Throughout, we used a **scrum-based software engineering** model.

Scrum is structured around daily standup meetings. The core idea is that anyone is invited, but team members rapidly talk through the progress they have made since the last meeting, the progress they intend to make before the next meeting, and any obstacles in their paths. We began using this model in a very formal way, but we soon had to adapt it to the realities of our project.

Our project did make it easier to follow scrum practice because it necessitated physical co-location. Our project uses ROS extensively (discussed in section 5). ROS requires a Linux Ubuntu operating system, and most of us had stability problems with ROS on our home machines. Moreover, Robot DE NIRO could not be taken out of the Robot Intelligence Lab without advisor supervision. All of this meant that when we worked on Project Fezzik, we did it while physically in the Lab, on shared Lab machines. When we were present together, this made communication, standups, and progress very easy. After all, it is much harder to make code comments over email or text than in person.

4.1.2 The Challenges of the First Half

On the other hand, the practical reality of our schedules over the five months of this project made traditional scrum practices more formidable. We are, and have been, full-time students. The classic implementation of Scrum, with its daily meetings and single-week sprints, seemed best-suited for when a software engineering team was working on their project full-time. We did not all have the same class schedules, and so even the times we could be in the Lab varied tremendously.

Consequently, we can divide our project neatly into two phases: first, from January through March, and second, from April through May. Late March saw the end of our day-to-day class lectures, coursework, and a number of exams. Before this, we came to the Lab whenever possible, in between or after classes, alongside whomever else on the team with whom we shared schedules. Wednesdays were our team in-person days, and we often made more progress then than in the preceding week.

Organisationally, we began with daily scrum meetings, expecting to make quick work of our components. The more challenges we encountered, the more we realised that we had very little to update on a day-to-day basis. We reduced that to three times per week, and then further one digital scrum over the Slack messaging service, and another in-person on Wednesdays. In these early months, progress was slow and sometimes frustrating. When we could not all be in the Lab, it seemed to make most sense to do as much independent work as possible. We geared our time to learning ROS and probing for component possibilities. Just trying to run a single ROS program without dependency or compilation errors felt like a ludicrous, insurmountable barrier.

Our original aim, following agile techniques, was to arrive at a primitive prototype by late February. This self-imposed deadline came and went without us close to a solution. Learning ROS, with its
complex concurrent communication, took far more time than we expected. Worse, each new component was less straightforward and less reliable than we planned for. However, most of all, we lacked consistent time in the Lab as a team. It was difficult to overcome our obstacles when we worked alone or in pairs. Since we were all new to ROS, we needed to learn as a collective.

The sprints we scheduled on a week-to-week basis were typically around testing new packages or building prototypes for our components. We worked hard just to see evidence of the data that DE NIRO’s sensors were supposedly processing.

4.1.3 The Progress of the Second Half

After our classes finished, the situation changed tremendously. By this point, we had all become much more adept with ROS. We also had significant time pressure – we were scheduled to present a working demonstration of Fezzik to the public at the Imperial Festival in late April. Most importantly, we could now work full-time in the Lab on consecutive days. It became far easier to build on earlier work, far easier to see the connections between breakthroughs in one component and another, and far easier to try to integrate the components into an increasingly complete whole.

The first three weeks of April were where our project really came together. Now that we were there daily, we could run daily scrum meetings consistently. They served their purpose well – we became more familiar with one another’s challenges. Arriving at a working method for grasping helped us establish just how precise our navigation needed to be so that DE NIRO could reach the objects successfully.

Other elements of scrum development – rapid iteration and frequent client feedback – were also in evidence. When our classes finished, so did Dr. Kormushev’s outside teaching obligations. He was in the Lab even more often. As soon as we made progress with a component, we could show him right away and get his feedback. This was very helpful. He would tell us when we were going down a fruitless path, saving us time. When we were at a dead end, as it happened very frequently, for example as we started with an unreliable navigation stack, he pointed us toward alternatives. At the start of our project, the fetch routine was our idea, and Dr. Kormushev enthusiastically supported it. With the simple domestic assistant in our minds, we could all visualise where we needed our project to go. Moreover, with the Imperial Festival coming up, Dr. Kormushev remained closely linked to our progress. Having built Robot DE NIRO himself, Dr. Kormushev regularly offered us ideas for extensions or simplifications.
When the Imperial Festival came, we were ready. It had its fair share of challenges – a noisy, real-world environment, robot interaction with eager children, and a wheelchair base battery that could only last so long – but it was still a great success. We had merged our components into a reasonably working whole, and we managed many successful demonstrations to the public. Our work was a highlight of the festival, being featured as one of the most memorable and loved interactions by the Imperial College London organisers.

After our exams finished in May, we were moving the fastest we ever had been. We were all now intimately familiar with the robot (having all had to troubleshoot each component countless times), and we knew what would be achievable in the time we had left. In our final few days, we made more progress than we had in a month of work earlier on, adding features (like audio input), improving existing functionality (giving DE NIRO the use of his left or right arms to grasp objects), and improving the modularity and design of our software.

Our final product represents an enormous amount of work, with an accelerating level of confidence and cohesion from us as a team. One reason this worked so smoothly by the end is that we had optimised the tools we used to organise ourselves and do our work. We explain our use of these tools in the next section.

4.1.4 Project Management Tools

We used many tools to track our tasks, mark completion or problems, plan our calendar, and communicate with one another.

Messaging Group messaging was our most heavily used communication tool. We used Slack for official discussions. Using Slack’s channels to subdivide conversation, we recorded all our weekly digital standup meetings from the first half of the project there. Because it was more commonly the way we communicated for casual purposes, we also tended to use WhatsApp for logistical purposes. Both WhatsApp and Slack have reliable file transfer capabilities, and we made use of these as well. We also used WhatsApp to keep open an ongoing group thread with Dr. Kormushev, on which he participated freely.

Official Communication We used web-based email and calendar applications for all official communication and scheduling. We used our given @doc.ic.ac.uk email addresses on all submitted reports, for direct communication with Professor Davison, and to receive all task assignments and updates through our project management tool, Asana. We also maintained and used a group mailing list (fezzik@imperial.ac.uk).

File Storage We used Google Drive as a cloud-based file storage provider. Although we connected this to our personal email addresses, and not our Imperial addresses, the increased convenience was well worth it. Google Drive gave us the added benefit of easily taking shared notes or co-editing scratch documents. We kept our in-person meeting minute notes here, for example. Meeting minutes were used during important meetings to log key decisions, advice from our supervisor and ensure aligned understanding of deadlines (See Appendix for snippet).

Task Scheduler Perhaps our most feature-rich tool was Asana. Asana is a web-based project management application where a team can track projects, assignments, and deadlines, adding tasks and generating reports as necessary. This was most helpful in getting organised early on, and we used a Kanban board method to mark what needed to be done in a given sprint.
Figure 7: The state of our Kanban boards showing our backlog, current sprint, and completed tasks near the end of our project.

4.2 Software Development

4.2.1 Git

We used Git [5] as our primary version control system. Working with multiple machines, sometimes without access to the Internet, a distributed version control system was our obvious choice. We hosted our Git repository with Gitlab.

As a team, we made sure to follow best practices for version control throughout the project. This meant we began a work session with a `git fetch` and a `git merge`, and we always ended with an informative `git commit` and a `git push`. We built our Git repository around our root ROS workspace, which we called `fezzik-project`. We describe the details of ROS in Section 5. For now, the main idea is that separate software modules (often aligned with separate components) live in independent subdirectories called packages.

Our already modular approach to the project made it easy to create separable feature branches. When a subcomponent was complete or needed to be used by some other component, we could merge back
in to master, but since these features typically resided in their own ROS packages, there were rarely complex merge conflicts.

4.2.2 Testing

Software development in robotics is unlike software engineering elsewhere. The smooth, seemingly frictionless execution of code gives way to the literal grinding of gears. We never expected factors like noise, inertia, light, and safety to matter, but each one became relevant to the way we designed, developed, and – especially – tested our code. Given the uncertainties and vagaries of a real-world application, it was imperative that our testing was continuous, rigorous, and evident. We typically used these three approaches:

1. Unit Testing
2. Coverage Testing
3. Uncertainty Testing

The first and second are important parts of any software project, and certainly of any software project developed with agile. They encompass code quality, and they enforce a particular software behaviour, both necessary. However, even after we wrote unit tests and ran coverage reports, it was clear that a major essence of our project was not being tested. We added the third approach as an essential and additional step, to test whether our robot would reliably and consistently perform even in a noisy, real-world environment.

**Unit Testing**  Having written the large majority of our code in Python, we used the unittest framework [6] for unit tests, which provides a standard set of assert operations, like `assertTrue()` and `assertEquals()`.

Our goal was to design tests with a white box framework. The reason for this is before experimenting with and trying out our components, we usually did not know how they could or would work. We felt we could not write tests until we knew what would be possible. Afterwards, we were able to write tests that thoroughly covered our functionality. The disadvantage, of course, is that we may have missed parts of our specification. However, because the fetch routine was such a straightforward idea, easily imagined in its domestic assistant capacity, we kept a good grasp on our requirements.

We tested our code extensively with unit tests. Our scripts typically contain one logical unit per file (often a class), placed in the relevant package’s src directory. All tests were placed in a peer test directory and for clarity, were given the same name as the file they test, but prepended in this way: `test_[FILE_NAME].py`. We ran each unit test against one particular aspect of the code. For example, we tested the state of the robots arms, the length of an array holding coordinates, and the accuracy of a mapping from target object to tag number. These unit tests are the smallest testable level in our code.

In total, we wrote 19 unit tests with each test comprising approximately 3 to 5 assert statements on average, with roughly 80 assert statements in total. Figure 8 shows one of these for the ObjectRecognitionServer class. The corresponding results are shown in Figure 9.

**Coverage Testing**  For this, we used the Coverage.py framework [7]. This mainly ensures that all our final code is actually functional, without extraneous, unused code. We produced coverage results of a full run of our final program. We include the full coverage report in the appendix, in Figure 31. In Figure 10 we show extract of the coverage results. We achieved an average coverage rate of 95.8% on a
server = ObjectRecognitionServer("obj_recog_server", ObjectRecognition)

class ObjectRecognitionServerTests(unittest.TestCase):

    def test_configs(self):
        self.assertEqual(type(server.config), dict)
        self.assertTrue(len(server.config) > 0)

    def test_median_points(self):
        client = ObjectRecognitionClient("obj_recog_server", ObjectRecognition)
        median_points = client.make_request()
        self.assertEqual(type(median_points), Point)

    def test_server_attributes_after_call(self):
        # Server contains array of data points for object (x, y, z) coordinates
        self.assertEqual(len(server.x), server.max_data_points)
        self.assertEqual(len(server.y), server.max_data_points)
        self.assertEqual(len(server.z), server.max_data_points)
        self.assertEqualFalse(server.take_measurements)

Figure 8: Unit tests for the ObjectRecognitionServer in object_recognition_server.py

Figure 9: Results for ObjectRecognitionServer unit tests.

total 1,362 statements. We only ran coverage tests on scripts we ourselves wrote with functional code. Almost all our missed statements are the result of special cases, such as an untriggered exception. Although these may rarely be called, they are critical to prevent malfunctions that pose the risk of product damage or worse, user harm.

Uncertainty Testing Tools  Some tests could not be an automated, constant part of our project. DE NIRO localises differently in different environments – like stark, empty corridors or in furniture-filled rooms. Where there was real-world uncertainty or inconsistency, we tested more manually. For these, we used a couple of tools that bear mentioning. RViz [8] is a multi-purpose 3D visualiser for ROS, able to show a variety of data. Most often, we used it to visualise navigation maps, Kinect point-clouds, coordinate transforms, and other 3D data.

We also used the log function in ROS, which we usually evoke with rospy.loginfo(STRING) or rospy.logdebug(STRING). However, ROS logged an enormous amount of information, and sometimes did so very rapidly (especially when logging published topics). We needed a way to filter and
To achieve what we needed, we created our own **graphical user interface (GUI)**, using it throughout our development process. We integrated RViz, ROS logs, and camera outputs, presenting them all through rqt [9], a Qt-based standard package for GUIs in ROS. In Figure 11, it is clear how useful this GUI was when debugging or testing.

<table>
<thead>
<tr>
<th>Package Name</th>
<th>File</th>
<th>Total Covered Statements</th>
<th>Missed</th>
<th>Coverage</th>
<th>Missing Lines (Code Line Number)</th>
<th>Reasons for missing statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>grasping</td>
<td>grasping_handler_server</td>
<td>118</td>
<td>3</td>
<td>97%</td>
<td>168-188</td>
<td>Special case: Can’t reach object</td>
</tr>
<tr>
<td>grasping</td>
<td>grasping_status_client</td>
<td>18</td>
<td>2</td>
<td>89%</td>
<td>28-29</td>
<td>Special case: If grasping status server fails</td>
</tr>
<tr>
<td>grasping</td>
<td>grasping_status_server</td>
<td>73</td>
<td>4</td>
<td>95%</td>
<td>98; 102; 107; 112</td>
<td>Special case: Arms not correctly enabled</td>
</tr>
<tr>
<td>grasping</td>
<td>offering_object_client</td>
<td>18</td>
<td>2</td>
<td>89%</td>
<td>26-27</td>
<td>Special case: Arms not untucked properly</td>
</tr>
<tr>
<td>grasping</td>
<td>offering_object_server</td>
<td>44</td>
<td>0</td>
<td>100%</td>
<td></td>
<td>Special case: If offering object server fails</td>
</tr>
</tbody>
</table>

Figure 10: A sample of coverage tests run on scripts in the grasping package.

Figure 11: Project Fezzik GUI, based on rqt
4.2.3 Documentation

Documentation is a prime example of how our tools reflected our growing maturity and expertise in Project Fezzik. Well-structured documentation from previous groups would have been a great help to us throughout this project. However, most previous documentation was based on simple word processing documents that were hard to decrypt: several people had been working on them in parallel.

Nevertheless, when we started our own project, we also began with a simple shared Google Doc. It simply did not appear to be worth the time to have a more complex structure, especially before we knew how we would build our project. As our Google Doc grew out of control – partly because it was so easy to dump into it – we decided to switch to \LaTeX{} to force more discipline.

As our project matured, we were exposed to more and more high quality online documentation. The best were a long way from a linear text document; they were, instead, interactive websites. In our next scrum meeting, we decided to do documentation right. We wanted to develop documentation that would not only be useful to the Lab, but that could also become the standard for any future documentation. The Lab may eventually choose to have one large website with several logically separated but interconnected subsections.

In the end, we decided to build our documentation with **Sphinx** [10], an industry standard documentation tool. Sphinx uses text files written in the lightweight reStructuredText markup language to compile and style content into a variety of formats, including HTML, \LaTeX{}, EPUB, XML, and more. Our documentation is permanently available at: [https://robot-intelligence-lab.gitlab.io/fezzik-docs/index.html](https://robot-intelligence-lab.gitlab.io/fezzik-docs/index.html). Figure 12 shows how it looks and works. We sincerely hope this will prove useful for other groups going forward.

![Figure 12: Our documentation is an interactive website with a menu on the left and structured guidance for each component and for the documentation itself.](image-url)
5 Robot Operating System

5.1 ROS Overview

ROS, as the textbook says, “is an open source framework for getting robots to do things.” [11] A component might find or return a value used by another component, but sharing resources is far from trivial. We needed a simple, consistent way to communicate across our project, optimised for robotics. ROS served this purpose. Alongside our state machine, ROS was our other primary, persistent tool to link our components.

ROS is not a true operating system: it neither uses a separate kernel nor shares kernel privileges with the host operating system. Instead, ROS serves as a middleware on the application level of the network stack. It abstracts concurrent execution and real-time networking among distributed components. Different components can be linked up to ROS separately, even in separate programming languages (though we stuck to Python). ROS’s modular and open source design also makes it feasible to test others’ packages.

Though ROS is a boon to robotics today, it is still quite young, having been developed about a decade ago [11]. It is not as robust and easy to use as, say, Python itself. ROS’s online resources, though helpful, are not as plentiful as would be ideal. Frequent releases means old incompatible packages are scattered across the robotics community. Our Kinect could interface only with the penultimate version of ROS, while our LIDAR preferred the most modern one. We ended up with multiple machines running multiple versions of Linux and ROS. ROS was not easy to learn, and we struggled with the early learning curve. Still, it was the best, most capable option, and we made extensive use of it.

5.2 File Structure

ROS collects all project files in a single directory, known as its workspace. User-built packages, which housed our components, themselves reside in the root’s src subdirectory. Packages then have their own src directories to house scripts. Using mandatory package.xml and CMakeLists.txt files, ROS can identify which packages are dependent on which others.

This file structure is a fragile latticework. Even one file out of place could crash the entire ROS instance. ROS’s default behaviour requires precise positioning of files in the correct directories, with corresponding manual updates to the package.xml and CMakeLists.txt files when the user needs to add new dependencies or features. It was some time before we could add and incorporate our own packages easily.

5.3 The ROS Master

ROS links software modules in a graph-based network. Any module – called a node in ROS – can communicate one to one with any other, or one to many with all others. The central hub allowing connects and logging which nodes are adjacent to which is the ROS Master [12]. To run any ROS program or use ROS commands, one must create a ROS Master, typically by executing roscore on the command line.

New nodes declare themselves by asking the ROS Master to add them to its registration table [13]. The ROS Master also starts the Parameter Server [14], which works like a globally defined Python
dictionary or a C++ unordered map. Any node can read or write to the Parameter Server to conveniently pass basic types of data.

The ROS Master may seem like a hub router, blocking or permitting message transmission. However, this is not the case. Nodes communicate to one another without the interference of the ROS Master, which acts more like a page table, managing node address locations. roscore also launches some services auto-establishing some nodes and assigning default network ports, simplifying our work there as well. One need only one ROS Master per project (only a single address table across linked components is sensible). For our project, we use the ROS Master initiated automatically by the Baxter robot arms of Robot DE NIRO [15]. Once a ROS Master is in place, and as long as any other necessary device is connected to the same network, we can plug in any new nodes.

Once connected, ROS commands work in scripts with the rospy library [16]. We declare nodes with rospy.init_node("NAME_OF_NODE") and can read (or write, if the key does not exist) from the Parameter Server with rospy.get_param(KEY, VALUE). To communicate from one node to another, ROS offers many methods, but we chose to use two, one that broadcasts information and one that responds to a request.

5.4 Asynchronous Communication: Publisher/Subscriber Model

The first is the Publisher/Subscriber model. Often in robotics, sensors need to continuously update a data field, like a coordinate transform to update positions. By publishing that information to all nodes, any component can access this information. First, a node chooses or establishes a topic on which it will communicate, declaring that it wishes to write to that node. Then, any other node can subscribe to that topic by indicated this to the ROS Master, and kicking off a callback function that usually awaits a particular value.

```
# Publisher declaration, with customisable data type and queue size
handler = rospy.Publisher("NAME_OF_TOPIC", String, queue_size=10)

# Subscriber declaration, with customisable data type and callback function
rospy.Subscriber("NAME_OF_TOPIC", String, callback)
```

Figure 13: Python code syntax for publisher and subscriber declarations; these lines would usually exist in separate packages.

We use publishers and subscribers extensively, but especially so in our navigation stack. When DE NIRO builds a navigation path to a goal, the robot selects a particular odometry configuration to lead it along that path. It publishes that information (including distance from its starting point) to a topic subscribed to by the wheelchair base, which in turn calculates how much power to send to the wheels. In most cases, we wrote these publishers and subscribers ourselves. Sometimes – as with this example – we modified an open source script.

5.5 Synchronous Communication: ROS Service Model

Sometimes, constant publishing is excessive. A variable that only toggles once may be better served by the Parameter Server. However, very often, we require a response to a specific input, or we need to pass a data of a type too complex for the Parameter Server. For instance, during object recognition, we must pass in the identifier of the requested object and return a tuple with the correct coordinates.
To mimic function behaviour across ROS, we use **ROS Services**. In a service, one package is the client, passing parameters through the ROS Master to another package. This other package is the server, which, upon receiving these parameters, processes them and (optionally) returns back a new value to the client. This is not as prosaic as it sounds – it means that any computing device connected to any other through ROS can communicate as in the same local stack. Implementation is a little more complicated. **CMakeLists.txt** and **package.xml** files need to be edited to reflect package dependencies, and a special ROS Service file and **srv** subdirectory needs to be added. The simple text file declaring input parameters and return values. ROS Services can support more complex types. The file is the entirety of our **GraspingHandler.srv** file, part of a service that takes in a 3D coordinate [17], runs a grasping action, and returns the outcome of that action:

```
geometrymsgs/Point object_position

uint8 grasping_status
```

The client and server must both invoke call and response methods. When the client imports the new **.srv** file, ROS treats it as a class declaration, setting the parameters and return values as publicly settable attributes. The client simply waits for a signal from the server, at which point ROS creates an instance of this class with a non-null return value. From here, the client accesses the output of the ROS Service by dot accessing the return attribute of the dynamically created object instance.

We used ROS Services when a unique value from one component must elicit unique behaviour in a different component. They also proved useful in verifying that a component was active. During the **InitRobot** state (see Section 2.2), we run a suite of ROS Service tests to verify that the packages can indeed interact. Inactive servers will never return a value to the client package, timing out.

### 5.6 Experimenting with ROS

ROS required us to learn, test, and apply a programming methodology entirely new to us all. At first, we had little sense of ROS’s capabilities. Over time, as these mechanisms became increasingly familiar, we applied them with more finesse. When we now choose to use, say, a ROS Service, it is usually after having tried and rejected a Publisher/Subscriber. Next, we discuss our components in detail, and it is clear that ROS is used throughout.
6 Component Features

In this section we explain how each component works. For each one, we tested various alternatives, sometimes pre-existing ROS packages and sometimes creations of our own. We explain the functionality of each component here, what trade-offs we had to consider, and how each one fits into the broader fetch routine. Since this project, unlike many software products, is built for real-world use, we also discuss some concomitant challenges.

6.1 Command Input (Owner: Clara Pouletty)

State 3 Listening

There is no point to a fetch routine if DE NIRO cannot accurately record the object to be fetched. Nevertheless, there are many ways to give this input. Users could type an input. Users could tap a key on a networked device. Users could show DE NIRO a picture of the desired object. User could point toward their desired object. A user could simply speak.

We considered each of these options, and we tested almost every one. With the idea of a domestic helper robot in mind, the ideal choice became obvious: The most natural, most human way to convey one’s desire is to voice it. From the beginning, this was our goal for input. In the course of our project, we worked with three APIs and implemented working service architectures: Amazon Alexa and Amazon Web Services (AWS), the Google Cloud speech-to-text functionality [18], and the CMUSphinx speech recognition library [10]. We accessed the latter two with the Python speech recognition interface [19].

Our original goal was to integrate the Amazon solution. This involves having an Amazon Echo wait to hear a predefined sentence structure, and then send a recording of that sentence to AWS, which would process the sentence to ultimately return a string as a response. At least, this is how it was to work in theory. Few component methods proved as frustrating as this one. An early challenge was more local – we had to negotiate with the university IT services to keep a WebSocket open in order to receive the inbound string. We were only allowed a single specific machine for this.

Then came the trouble with Alexa itself. We tried three different integrations, each one less successful than the last. The most effective method required using an out-of-date Amazon library, and guaranteed that the first five audio commands would – mystifyingly – result in no returned information, after which, we could retrieve strings with reasonable accuracy. However, being locked to a single machine significantly restricted the functionality to within the Lab itself. We needed to seek alternatives.

Luckily, we had already been testing the Google API client. This was easier to integrate and yielded the highest performance in multiple test runs in the Lab, even with arbitrary word inputs. However, this API was very vulnerable to ambient noise, and it certainly was not usable for the Imperial Festival. It could still work for an at-home deployment, but even there, a more resilient translator would be ideal.

Both of these APIs required internet access. Although we could guarantee a stable internet connection in the Lab, we could not in the wilder world outside. Plus, even with WiFi access to the internet, the connection could potentially interfere with the LAN WiFi broadcasts of various ROS topics. An offline speech recogniser, even if more limited in vocabulary, might be preferable.

Offline processing, “designed specifically for low-resource platforms”, is exactly what CMUSphinx of-
fers \cite{20}. In its default configuration, CMUSphinx’s recognition accuracy is notably worse than the online counterparts. It failed more often than not, getting worse with background noise. We could not use CMUSphinx as it was. Because CMUSphinx and the Google API were both part of the Python speech recognition interface, they share a superset of features. Among these is a feature that listens to background noise and calibrates the recognition system to ignore ambient sounds. This improved the performance of both packages in noisy environments.

Despite this improvement, CMUSphinx still struggled with comparing incoming audio files against the entire language possible space. To simplify its processing, we introduced a feature called JSpeech Grammar (JSG) to CMUSphinx. JSG adds a grammar to map any words heard into its expected format, using regular expression-like notation. We used this to define a short logical command sequence.

```plaintext
public <fetch> = [ <command> <article> <object> ] [ enumerate ] ;

<command> = fetch me | give me;
<article> = a | an | the ;
<object> = coffee | water | apple juice | football ;
<enumerate> = enumerate
```

Figure 16: Our markup of a simple fetch command syntax to reduce CMUSphinx’s possibility space.

This grammar restricts recognisable sentences to a pre-defined set of word sequences. This limits the generalisability of the speech recognition service for this particular API, but it is a realistic, non-limiting condition for our personal assistant robot setting. Increasing the number of possible commands lowered the recognition rate slightly, and we therefore kept the grammar definition simple for our final product.

When considering how to increase DE NIRO’s openness to inputs of multiple types, and with the Imperial Festival on our minds, we came back to the problem of very noisy environments. With indecipherable audio cues, perhaps visual inputs could be a replacement. As an alternative solution, we built a command input option that uses the same ROS markers we discuss in Section 6.4 with the Object Signs we introduce in Section 6.7. When DE NIRO cannot listen, it can usually still see. By showing DE NIRO markers of what we request, DE NIRO can convert those markers directly to a saved object preference. This worked so well that it is a simple toggle in our final report to switch from audio to visual input.

6.2 Audio Output (Owner: Nico Smuts)

| State 1 InitRobot |
| State 2 Idling |
| State 3 Listening |
| State 4 RememberingUser |
| State 5 Navigating |
| State 6 Grasping |
| State 7 Returning |
| State 8 SeekingUser |
| State 9 OfferingObject |

Audio Output made an extraordinary impact. DE NIRO is specifically named for natural inter-
action, and hearing it “speak” immediately led to more relaxed interactions from users. The added
bonus is that verbal output serves as a debugging tool. The particular audio output package we chose
consists of a ROS server using the eSpeak Python library \cite{21, 22} to verbalise any string sent to it,
which is then output through DE NIROs loudspeakers.

Audio Output is unique among the components in that it is called in every state. Upon entry, the state
machine triggers Audio Output to keep users updated on DE NIROs current status or next objective.
Sample verbal outputs include “Thank you [name of person identified via face recognition], I will now
fetch you [object]” and “I have failed to fetch the [object]. I will now return to base.”

We elected to use eSpeak over other, sometimes more sophisticated speech packages, including Amazon
\cite{23} and Google’s \cite{24} text to speech APIs. The key advantages of eSpeak are that it requires few
extra scripts, needs no internet connection, uses little processing power, and exhibits no noticeable
lag between when an instruction is sent and when the verbal output is produced. We designed the
audio output package to operate concurrently with other components, whereas the individual text to
speech instructions are processed in series. This allows DE NIRO to provide verbal commentary while
grasping and navigating, while eschewing overlapping speech when speech commands are sent in quick
succession.

6.3 Mapping and Navigation (Owner: Nico Smuts)

\begin{align*}
\text{State 5} & \quad \text{Navigating} \\
\text{State 7} & \quad \text{Returning}
\end{align*}

Navigation is a particularly involved feature of our project. For DE NIRO to navigate successfully
through physical space, it must:
\begin{itemize}
  \item map its environment,
  \item locate itself within that environment, and
  \item plan a collision-avoiding route through that map.
\end{itemize}

These steps form a control loop and are referred to respectively as mapping, localisation, and trajectory
planning.

\textbf{Mapping} \quad We map a location before running the fetch routine to produce a static bitmap image.
To do this, we use the ROS hector\_mapping \cite{25} package, relying on DE NIRO’s LIDAR sensor to
generate a map of a given location’s walls and static obstacles. The LIDAR sends a wide, 270-degree
scanning plane at about a third of a meter from the ground, detecting any objects as 2D artefacts. hec-
tor\_mapping uses Simultaneous Localisation And Mapping (SLAM) to allow a robot to move through
a static map, continuously updating its position relative to that map.

Figure \ref{fig:static_map} shows a sample static map. Because DE NIRO uses this map as a reference, we can easily
create synthetic barriers on the map by simply adding darkened lines in the image (see Figure \ref{fig:static_map2}). This
is very convenient when DE NIRO must remain inside a predefined section of open floorspace.

\textbf{Localisation} \quad The next step, localisation, occurs during the fetch routine. DE NIRO compares what
its LIDAR sees in real-time with the static map to estimate location and orientation, refreshing it
with LIDAR data at 40 Hz. When the LIDAR detects new objects, DE NIRO treats them as dynamic
obstacles and overlays on the static map. Too many of these and DE NIRO becomes paralysed.
Therefore, it is important for there to be distinct, asymmetrical, static objects in the scene so DE
NIRO does not get confused between one featureless wall and another. We only discovered this after many failed trials, when DE NIRO thought itself to be in distant, but similarly shaped, parts of the map.

**Trajectory Planning** Trajectory planning adds two more layers. The first, a “costmap,” adds a virtual cushion around static obstacles to reduce collision risk. The fixed virtual hexagon buffer around the body of the wheelchair base is yet another safety feature. Figure 19 shows the preceding map with a 10 cm costmap overlay around all obstacles. The second layer is an optimal path from origin to destination, determined by the ROS `teb_local_planner` navigation package.

`teb_local_planner` uses a “timed elastic band” approach [27], conceiving of path planning as a multi-objective optimization problem. It seeks to minimise the cost assigned to variables like total travel time and obstacle proximity. `teb_local_planner` updates an optimal path in real time to avoid static and dynamic obstacles (see Figure 20). The optimal path (in green) is broken out as a sequence of velocities (in red) required to reach the destination.

Having built an optimal path, the package publishes these velocities as floating point tuples containing linear and angular velocities. A velocity converter, building upon earlier Lab research [28], subscribes to these velocities, scales and smooths them, and finally republishes the new velocities as integer tuples. These instructions feed into a customised Mbed microcontroller developed by the Lab, which sends the necessary current to each motor, thus producing the desired movement.

Figure 20: The original optimal path (left), encountering a dynamic obstacle (middle), adjusts in response (right).
The state machine calls the navigation stack via a ROS Service. The navigation server maintains current pose and mapping information and predefined destinations (as 2D location coordinates and a target yaw). Requests to move to these destinations are sent as simple string arguments, such as `warehouse` or `home`. The navigation server responds with a confirmation message and, after attempting to execute the instruction, returns the outcome of that attempt as a boolean value.

Navigation is a perfect example of a component that needed some manual, real-world tests to achieve reliability in noisy environments. We ran a variety of tests using a separate autonomous wheelchair base (Figure 21), identical to that of DE NIRO. This sped up and simplified our development process. Some of the manual tests we ran include:

- executing predefined blind movements,
- mapping an isolated room,
- mapping a room with dynamic obstacles,
- localising within a static map,
- updating localisation in real time,
- sending one navigation goal,
- sending multiple navigation goals,
- adding static obstacles mid-navigation,
- adding dynamic obstacles mid-navigation, and
- measuring displacement from ideal endpoints.

These led to several software enhancements, such as smoother velocity instructions and more robust localisation. This is where we learned of the need for uniquely shaped physical markers around the environment. We implemented a significant redesign after tests indicated that our initial trajectory planner (dwa_local_planner) [29], made a subtle but problematic decision. It first optimised for the desired X and Y coordinates (moving to the right location) and only then twisted to achieve the desired Yaw. teb_local_planner, which implements more natural, “car-like” movement via Ackermann steering [30], avoided these problems.

6.4 Object Recognition (Owner: Kim Rants)

<table>
<thead>
<tr>
<th>State 3</th>
<th>Listening (optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 6</td>
<td>Grasping</td>
</tr>
</tbody>
</table>

For object recognition, we tested many methodologies – each with various levels of success. Our goal was first, to recognise the correct object and next, to retrieve the coordinates of that object in 3D space. Both of these criteria needed to be robust and flexible. If, for example, the grippers did not work for a particular object, it would not matter if we could recognise that object accurately. Thinking again of the ideal of the domestic assistant robot, operating in a noisy, dynamic environment, we knew we needed to be able to swap objects easily and quickly.

Table 1 shows what we tested and how well those methodologies worked. Starting from the bottom, training our own model with OpenCV Haar Cascade [35] proved to be time-consuming and unreliable.
Table 1: Object recognition methodologies we attempted. Red is unsuccessful, yellow is potentially successful but with major drawbacks, and green is our chosen method.

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Ability</th>
<th>Tests</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROS Marker Chili Tags [31]</td>
<td>2D Fiducial Marker Recognition</td>
<td>Live tests</td>
<td>Very high accuracy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Distance tests</td>
<td>Very flexible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accuracy tests</td>
<td>Easy to retrieve coordinates</td>
</tr>
<tr>
<td>find_object_2d [32]</td>
<td>2D Object Recognition</td>
<td>Several objects in lab</td>
<td>Low accuracy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Distance tests</td>
<td>Very flexible</td>
</tr>
<tr>
<td>visp_auto_tracker [33]</td>
<td>2D Fiducial Marker Recognition</td>
<td>Different size markers</td>
<td>High accuracy up to 2 meters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Distance tests</td>
<td>Needs large fiducial markers</td>
</tr>
<tr>
<td>Tabletop [34]</td>
<td>3D Object Recognition</td>
<td>Never ran properly</td>
<td>Depth/point-clouds (Kinect)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Could not pass on data</td>
</tr>
<tr>
<td>OpenCV Haar Cascade [35]</td>
<td>2D Object Recognition</td>
<td>3 objects (stored videos)</td>
<td>Very time-consuming to train</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 objects (live videos)</td>
<td>Limited flexibility</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Limited coordinate precision</td>
</tr>
</tbody>
</table>

Worst of all, it never generated precise coordinates. Tabletop [34] seemed promising at first but turned out to be one of the most painful sunk costs of our project. It always seemed on the cusp of working, because we could see the point clouds it generated. It was also more ambitious, seeking to identify 3D objects consistently. Unfortunately, we never got it to work. visp_auto_tracker [33] uses 2D fiducial markers, and although this worked, it only did so at close distances. find_object_2D [32] uses edge detection to identify objects. It was therefore very flexible, though not very accurate.

In the end, we decided to go with ROS marker chili tags [31]. This package was very robust on integration and, critically, had near-perfect accuracy, even at relatively long distances. Also, by editing the launch files source code, we could easily adjust the physical size of the markers and still get reliable results. This was very effective. With chili tags, DE NIRO detects small fiducial markers attached to the target objects instead of the objects themselves. Because the tags are black and white and easily distinguishable (similar to QR codes), accuracy is extremely high. We were often surprised by how easily the camera could recognise markers that we had left lying on tables in the distance.
The package also easily returns the coordinates of the marker in Kinect camera’s view space. The coordinates are three dimensional (X, Y, and Z floats). To move the arms toward these coordinates, we had to convert into the arms’ view space and then send the coordinates to the grasping package through ROS.

We accomplished this by inverting two axes, with good old measuring tape, and by trial and error. The Kinect was simply placed on top of DE NIRO – it was up to us to calibrate exactly what the right transform was between the coordinate system origins. This worked quite well.

In some very infrequent cases, we observed an outlying spike for a single coordinate, as if an axis were out of sync. We hypothesise that this is due to lighting differences, where the Kinect gets momentarily confused by a harsh reflection on one end or a highly matte surface on the other. Unfortunately, these problems sometimes appear randomly, like “Heisenbugs,” and they are difficult to reproduce. Practically, we solved this by recording an array of 3D marker coordinates – 20 in total – and selecting the median. Since these spikes would normally only happen once and never consecutively, it would probably have been sufficient to collect just 3 coordinate sets and then take the median, but collecting 20 did not take much longer and is more robust. Introducing the median technique helped make DE NIRO more resilient to camera confusion.

6.5 Grasping (Owner: John Lingi)

State 6 Grasping
State 9 OfferingObject

After identifying the correct object from the warehouse of objects, the next step is to physically retrieve it. As we came to learn, this is a non-trivial task in robotics. We did not originally prioritise grasping as a core skill. We hoped to use an off-the-shelf grasping package or perhaps use some built-in intelligence in the Baxter robot arms to memorise movements. However, after we started looking into this problem midway through our project schedule, we realised how little we actually knew. There were, in fact, many different ways we could use grasping to create a climactic, even delightful, moment in our routine.

At the highest level, grasping involves manipulating the arm joints and, separately, manipulating the grippers. DE NIRO’s arms have seven joints, so when receiving 3D Cartesian coordinates from Object Recognition, determining where each joint should go, at what speed, and in what order is quite involved.

We solved this by using a ROS API for the IK Solver Service [37]. This service finds solutions for the joint angle problem, given the input of a particular coordinate goal. Still, using this left many practical issues. The IK Solver gave the ideal end joint angles. But it did not promise an elegant path from DE NIRO’s current arm pose to those final angles. We experienced this first hand several times as DE NIRO would knock over the objects or even attempt to flip the table holding our objects.

Presented with this problem, we came up with two possibilities. First, we could try to modify the IK Solver itself to add fixed constraints to angle possibilities, tightening DE NIRO’s zone of movement. We felt this was a less robust and potentially time-consuming fix. We chose the simpler second option. Here, we simply added an intermediate point 15 cm before the target object marker. This gave DE NIRO the freedom to move its seven joints freely up until a fixed point in the YZ plane of the target object. From there, it moved in a straight line in the X (depth) direction.
Having reached the correct position, the next task is to grip the object. We fixed the grippers orientation to avoid grip attempts from upside down or other odd angles, reducing the joint angles to solve to 6. The Baxter arms have a noticeable error zone due to lack of precision, and they often will gently knock objects out of place on their way towards them. Cylinder-shaped objects with heavy foundations were best to grasp and least likely to fall over. For demonstration purposes, we used bottles of juice.

At first, after grasping the object, we moved DE NIRO straight along to the next state. However, after all the build-up, this felt cathartic. To sweeten that instant, we added the “Hero Mode”. Before returning home, DE NIRO raises an arm to the sky, object in hand, gleaming with pride. Observers, especially at the Imperial Festival, seemed to love this display of panache. It made people ask to see DE NIRO try again.

Hero Mode also become quite practical. After returning back to the person who gave the initial instruction, DE NIRO only lowers its arm if the correct user is found. It became an extra test of accuracy.

Finally, in OfferingObject, DE NIRO transitions its arms to “Handover Mode.” We designed Handover Mode with a smooth user interaction so even small children can seamlessly take the object from DE NIRO’s hand with minimal force. We accomplish this by moving DE NIRO’s arm to a flat horizontal state. Then, in a spinning loop, we query the coordinates of the hand position. If the hand position changes in the Z direction (i.e. the closed hand is tugged up or down), then we release the grippers. It is as if DE NIRO hands over the object on a gentle pull by the user.

As a final addition, inspired by Boston Dynamics, we decided to implement more realistic awareness. If DE NIRO is partway toward grasping an object, but someone teases DE NIRO by moving it out of its way, the robot will react. DE NIRO checks whether its grippers fully close. If they do, it implies DE NIRO had grasped no object. If fully closed – meaning empty – DE NIRO will seek the object location once more and try to reach for it again. Furthermore, we started with only using DE NIRO’s right arm. We recently activated DE NIRO’s left arm as well, allowing the robot to naturally choose the left or right arm depending on the location of the object relative to DE NIRO’s own body.

Grasping began as a more mechanical routine and has progressed to become far more natural and less
linear. Originally, the grasping function would return a simple boolean to represent the success of the grasp attempt. Upon failure, DE NIRO just reverted to its starting position. Now, we have added an extra outcome to capture the case where DE NIRO was on the right track to grasp an object, but some externality led it astray. If it fails, DE NIRO reacts in a way dependent on the cause of that failure and will try again if sensible. We illustrate our complete grasping process in Figure 27.

Figure 27: Illustration of the grasping phases.

6.6 Face Recognition (Owner: Fabian Falck)

We deploy Face Recognition to detect and recognise a human face. A domestic assistant robot, we felt, ought to remember the person who initially requests an object. That requester should receive the requested object, and not anyone else. We use Face Recognition early on in the RememberingUser state (when hearing a command), and later in the SeekingUser state (when returning the requested object).

In a first iteration, we built a face detection component, assuming that its functionality was sufficient for our purpose. This was a clear graceful downgrade – simply detecting a face (but not knowing who it was) would be of benefit if recognition were not possible. Still, we hoped to reach the ideal form.

We began to use OpenCV [38], one of the most popular computer vision libraries. OpenCV is very well documented for Python. With about 20 lines of code, we could detect faces very accurately with just a regular webcam. At its core, the package we used from OpenCV uses the Haar Cascade classifier, a feature based machine learning method. Haar Cascade combines multiple classifiers that have been trained on features extracted from the positive and negative samples of a dataset. It is good at finding
edges, lines, or rectangles in a cascade of classifiers. These strengths make it, without a doubt, a high-performing and stable face detection tool, though here again, it came short in differentiating among human faces.

Our next attempt was with the Python interface of the DLib C++ library. DLib’s underlying machine learning model is based on the Residual Learning for Image Recognition (ResNet) approach, using a pure convolutional neural network architecture. The pre-trained ResNet model has reached 99.38% accuracy on a standard benchmark and is seen as state-of-the-art as recently as February 2017. This model is more suitable to our task than the Haar Cascade model due to its very low rate of false positives. In other words, ResNet rarely states a particular face is present when a different or even no face actually exists. The difference is evident in Figure 28.

For DE NIRO to identify faces, the robot must compare its camera feed with pictures of faces in its database. In other words, this component works for people the robot already knows. In order to extract a feature representation — a necessary step to identify faces — we have to engage a precise execution of steps. First, the Kinect must retrieve and publish ROS image messages on a ROS topic at a rate of roughly 50 Hz. Then, with ROS’s CvBridge tool, we transform each image frame into an OpenCV image data type. Per frame, ResNet extracts image samples and then 128-dimensional vector encodings.

ResNet compares vector encodings across all the samples of this image frame with vector encodings from photos of the already known faces. By finding the Euclidean distance between testing and training encodings, ResNet can see how close the new image is to the remembered one. When this distance is below a threshold, ResNet will predict the incoming face as the remembered person and will mark the position of that sample with a bounding box. We show the final result on the right half of Figure 28.

Despite this success, there were multiple limitations. First, under very dark or bright lighting, the face tracking and recognition degrades significantly. Second, the distance of the faces to the camera must resemble the distance to faces on the training images. Third, when turning the head around the z-axis (yaw of the face > ± 45 deg), the face gets lost entirely. One solution might be to extract more samples before calculating distance to the known images. However, real-time processing is required here, and double the samples means double the processing time. As a compromise, we chose a sample rate so that the processed video stream has a slightly higher latency, but with an acceptable delay.

Figure 28: Haar Cascade for face detection (left) and ResNet for face recognition (right)
The drawbacks above reveal something about the nature of our project, notable in more than one of our components. While some research questions seem to be “solved” in highly constrained settings (as we see here with the need for optimal lighting, face distance, and head angle), performance drops significantly for real-world applications, like this robotics project.

### 6.7 Uncertainties

The ability and need to manoeuvre in changing physical environments is what separates robotics from dedicated software systems and other automation [44]. For a personal assistant robot, this ability is very evident. For example, the robot needs to be able to identify and grasp *different* objects, receive orders from individual with *varying* looks and voices, and navigate within *distinct* worlds.

It was very often the case in our project that a specific component would work in isolation, but then appear vulnerable when run in parallel to other components or in new environments. We experienced this first hand during the Imperial Festival. Initially, our idea was only to receive commands via audio, but audio input is very vulnerable to loud noises as well as to multiple and concurrent voices. As a backup, we chose to develop a second way to give DE NIRO commands: Object Signs.

![Figure 29: Three objects next to their related “Object Signs.” We used these during Imperial Festival to ensure a more robust command input state.](image)

Although these worked very well for the Imperial Festival, neither of the two methods (audio or signs) were without real-world uncertainties. Both type of commands come from humans, which in itself carries some risk. Different people will pronounce the objects differently, or potentially hold the Object Signs at an odd angle or simply move them too fast. Therefore, although it normally works well, the act of receiving instructions is a fragile process.

As DE NIRO navigates to a new location, we found it can get confused and freeze if there is too much movement. Since the LIDAR senses objects at about shin height, the problem mostly arises from moving legs. On the first day of the Imperial Festival, with the crowds and restless children, this was a major challenge. On the second day, we overcame this challenge by building low artificial walls and reducing DE NIRO’s distance to the object warehouse.

The real-world uncertainties are unpredictable in all directions. Object Recognition, to our surprise, caused few problems after we switched to the very robust ROS chili tags. Face Recognition, on the other hand, remained difficult, influenced by light conditions and face angle. Both represent uncertainties, especially when working with new crowds unaware of DE NIROs weaknesses. But even if lighting degrades Face Recognition, the fact that it exhibits few false positives makes the uncertainty easier to handle. Practically, by experimenting with its surroundings, DE NIRO *will* eventually
recognise faces in front of him and will very rarely guess incorrectly.

Once the correct object is recognised, grasping it carries its own idiosyncratic uncertainties. Most notably, the Baxter arm has major limitations in precision, since the platform is geared towards being a collaborative rather than an industry precision robot. This means that DE NIRO will sometimes gently nudge the objects when reaching for them. Regardless of how soft these pushes or catches might be, it limits the real-world conditions DE NIRO can handle at the moment. Too often, DE NIRO simply knocks over lightweight and narrow objects.

During the entire process, DE NIRO will speak its mind and inform the audience of its current state. We use eSpeak, which in general is a reliable and robust package. Nevertheless, we have had to do a fair bit of voice calibration to ensure that eSpeak will pronounce the words clearly enough for all audiences to understand. We tried to mitigate this by carefully restricting DE NIROs vocabulary.

All non-stationary robots are subject to changing physical environments, increasing the unpredictability of their behaviour. DE NIRO is no exception. As a software development team, we cannot avoid uncertainties entirely. We can only try to reduce them by understanding every process, running several edge-case tests, and restricting the robot’s excesses.
7 Final Product

7.1 Areas for Further Research

Although we have achieved a lot, there are still many exciting opportunities for further research with Robot DE NIRO. We highlight major opportunities here and refer to our documentation for an elaborated list.

1. Our navigation stack requires DE NIRO to start from a particular position on the static map, or else, it gets lost. Universal localisation would be a stark improvement, giving DE NIRO the ability to locate itself from any starting position on a static map.

2. Despite our work on the grasping task, DE NIRO can sometimes choose very inelegant or inefficient paths to an object. A prudent set of constraints on the IK Solver Service could moderate DE NIRO’s movements.

3. Audio (both input and output) could be more precisely tuned. An obvious area might be to increase the breadth of language DE NIRO understands. However, we think that a stronger ability to sift for relevant auditory cues amid literal noisiness would be especially helpful. Synthesised voices could be made more realistic, and the choice among verbalised strings more sophisticated.

4. Object recognition works extremely well with ROS markers. Future projects may try for reliable recognition and coordinate extraction without markers. This, of course, is a trade-off between versatility and reliability, but it is one that could bear fruit.

7.2 Project Fezzik

We have discussed design choices, outlined project management, introduced ROS, and portrayed each component in detail. At this point, we hope to synthesise all this information and holistically understand our final product.

When Robot DE NIRO is initialised, a ROS Master spawns, new nodes link to it, and, arriving at InitRobot, a sequence of status checks runs. We use our own GUI to ensure everything launches as planned. DE NIRO shifts to Idling, where it will sleep until activated. Activation sends it to Listening, where it awaits a keyword-laden fetch request. DE NIRO’s perception is very evident here. It listens for keywords, moving swiftly to RememberingUser, where it captures the person in front of it. At this moment, natural language and advanced image processing packages are both in use.

Armed with an instruction, DE NIRO starts Navigating to the warehouse of objects. Along the way, it constantly localises itself by comparing the LIDAR scan in real-time to its remembered static map. Based on these, DE NIRO plans its trajectory. Upon arrival at the warehouse, DE NIRO moves to Grasping. Here, DE NIRO performs the dual tasks of looking for a target marker and sending its arms to grasp at those coordinates. If it fails, DE NIRO may understand why it failed and re-attempt if sensible.

Object successfully grasped, DE NIRO raises its arm into Hero Mode, transitioning to Returning as it navigates back to its origin. Nearly done, DE NIRO shifts to SeekingUser, where it looks for matches to the requester. If successful, DE NIRO moves to OfferingObject, lowering its arm and holding it out in Handover Mode. Task complete, DE NIRO returns to Idling. All along this entire process, DE NIRO will speak its mind and inform the user of its current priorities and intended actions.
With these few steps, anyone can play fetch with our robot. We chose fetch because it is simple. Gauging success or failure is as simple as glancing at DE NIRO’s hands. However, we also chose it because it seems to be the backbone of many more sophisticated robot interactions. If a robot cannot understand speech, move, pick things up, or identify people, it is restricted from much more than just fetch. So, what we have strived to do is treat our work as the beginning of something more. We built Project Fezzik to be modular, easily remixed, and open to the imaginations of future designers and developers.

Today, Robot DE NIRO brings you coffee. Tomorrow, the world.
8 Acknowledgements

This work would not have been possible without the help of so many people across Imperial College London.

The Department of Computing is our home here. The Department was generous in allowing us to select a robotics topic in a lab outside the department, and the teaching and administrative staff provided us constant support throughout. Thanks in particular to Andrew Davison, our DoC supervisor, whose work first inspired us to seek out a robotics project and whose guidance led us to this one. We also want to thank Fidelis Perkonigg, our teacher and the organiser of this module as well as our MSc peers, who supported us and always gave us ideas.

The roboticists working at the Robot Intelligence Lab are a special bunch, and we learned a tremendous amount in their company. In particular, doctoral students Ke Wang and Roni Permana Saputra never hesitated to offer us hours of their time to help us answer questions. Our biggest thanks, of course, go to Petar Kormushev, a peerless engineer, an enthusiastic teacher, and a steadfast mentor to us on our journey through Project Fezzik. Petar’s encouragement pushed us to develop a project worthy of showcasing at the 2018 Imperial Festival, three weeks earlier than our official deadline. He threw his doors wide open to us and let us play fetch with his very own creation. Thank you.
References


A Appendix

Imperial Festival

The Imperial Festival allowed us to demonstrate the fetch routine to the public. It also served to test the resilience of both the software and hardware components of the project in a demanding, real-world setting.

Figure 30: The Imperial Festival demonstration included all key steps of the Fetch routine, namely command input, navigation, grasping, and offering the object to the user.
Design Supplements

Coverage Test Results

<table>
<thead>
<tr>
<th>Package Name</th>
<th>File</th>
<th>Total Covered Statements</th>
<th>Missed</th>
<th>Coverage</th>
<th>Missing Lines (Code Line Number)</th>
<th>Reasons for missing statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>audio</td>
<td>audio_out_client</td>
<td>24</td>
<td>2</td>
<td>99%</td>
<td>37-38</td>
<td>Special case: if audio server fails</td>
</tr>
<tr>
<td>audio</td>
<td>audio_out_server</td>
<td>23</td>
<td>0</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>audio</td>
<td>audio_status_server</td>
<td>35</td>
<td>3</td>
<td>91%</td>
<td>34-53-54</td>
<td>Special case: if files don’t exist</td>
</tr>
<tr>
<td>audio</td>
<td>talkback</td>
<td>30</td>
<td>0</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brain</td>
<td>initialise_parameters</td>
<td>18</td>
<td>0</td>
<td>100%</td>
<td>32-33</td>
<td>Special case: if status check fails</td>
</tr>
<tr>
<td>brain</td>
<td>status_check_client</td>
<td>20</td>
<td>2</td>
<td>98%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brain</td>
<td>sm_mach</td>
<td>27</td>
<td>0</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brain</td>
<td>sm_main</td>
<td>23</td>
<td>0</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brain</td>
<td>sm_status</td>
<td>164</td>
<td>5</td>
<td>95%</td>
<td>87-90</td>
<td>Special case: Status check fails</td>
</tr>
<tr>
<td>face_recog</td>
<td>face_sub</td>
<td>91</td>
<td>4</td>
<td>96%</td>
<td>86-97; 189-190</td>
<td>Special case: Exceptions not triggered</td>
</tr>
<tr>
<td>face_recog</td>
<td>face_sub</td>
<td>27</td>
<td>0</td>
<td>100%</td>
<td>41-42</td>
<td></td>
</tr>
<tr>
<td>grasping</td>
<td>get_joint_angles</td>
<td>7</td>
<td>0</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grasping</td>
<td>grasping_handler_client</td>
<td>59</td>
<td>8</td>
<td>86%</td>
<td>40-41</td>
<td>Special case: Only one arm sequence selected</td>
</tr>
<tr>
<td>grasping</td>
<td>grasping_status_server</td>
<td>38</td>
<td>2</td>
<td>98%</td>
<td>60</td>
<td>Special case: Object position not known</td>
</tr>
<tr>
<td>grasping</td>
<td>grasping_status_server</td>
<td>73</td>
<td>4</td>
<td>95%</td>
<td>94</td>
<td>Special case: Object not grasped</td>
</tr>
<tr>
<td>grasping</td>
<td>offering_object_client</td>
<td>18</td>
<td>2</td>
<td>89%</td>
<td>168-188</td>
<td>Special case: Can’t reach object</td>
</tr>
<tr>
<td>grasping</td>
<td>offering_object_server</td>
<td>44</td>
<td>0</td>
<td>100%</td>
<td>28-29</td>
<td>Special case: If grasping status server fails</td>
</tr>
<tr>
<td>mapping_2d</td>
<td>odometry_publisher</td>
<td>48</td>
<td>1</td>
<td>98%</td>
<td>98</td>
<td>Special case: Difference equal zero</td>
</tr>
<tr>
<td>navigation</td>
<td>navigation_handler_client</td>
<td>20</td>
<td>2</td>
<td>90%</td>
<td>29-30</td>
<td>Special case: Navigation server response fails</td>
</tr>
<tr>
<td>navigation</td>
<td>navigation_handler_server</td>
<td>52</td>
<td>3</td>
<td>94%</td>
<td>106-108</td>
<td>Special case: Action server not available</td>
</tr>
<tr>
<td>navigation</td>
<td>odometry_publisher_cmd_vel</td>
<td>27</td>
<td>0</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>object_recognition</td>
<td>depth_marker</td>
<td>40</td>
<td>0</td>
<td>100%</td>
<td>24-25</td>
<td>Special case: If object recognition server fails</td>
</tr>
<tr>
<td>object_recognition</td>
<td>object_recognition_client</td>
<td>18</td>
<td>2</td>
<td>89%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>object_recognition</td>
<td>object_recognition_server</td>
<td>54</td>
<td>0</td>
<td>100%</td>
<td>34-46</td>
<td>Special case: Recognised tags not exactly one</td>
</tr>
<tr>
<td>object_recognition</td>
<td>receive_instruction</td>
<td>37</td>
<td>2</td>
<td>95%</td>
<td>90</td>
<td>Special case: No server response</td>
</tr>
<tr>
<td>ros_comms</td>
<td>ros_client</td>
<td>42</td>
<td>3</td>
<td>93%</td>
<td>36-37</td>
<td>Special case: ROS exception in service response</td>
</tr>
<tr>
<td>ros_comms</td>
<td>ros_server</td>
<td>9</td>
<td>0</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ros_comms</td>
<td>ros_pdb</td>
<td>15</td>
<td>0</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ros_comms</td>
<td>ros_sub</td>
<td>9</td>
<td>0</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>speech_recog</td>
<td>speech_server</td>
<td>89</td>
<td>17</td>
<td>81%</td>
<td>141-146</td>
<td>Special case: No object recognised</td>
</tr>
<tr>
<td>speech_recog</td>
<td>speech_client</td>
<td>20</td>
<td>0</td>
<td>100%</td>
<td>153-155</td>
<td>Special case: More than one object recognised</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>189-192</td>
<td>Special case: API error and Unknown Value error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>204-205</td>
<td>Special case: Not understood after max attempts</td>
</tr>
</tbody>
</table>

Figure 31: Coverage tests for all of Project Fezzik.

Here, we show the full set of coverage tests for scripts across the project. Each test is accompanied by a line reference to uncovered code and a reason for these statements being missed. In total, there were 1362 tests with an average of 95.8% coverage.
UML Class Diagram

This UML class diagram shows the main classes of the components in our code base. We display only the key component classes with their most important attributes and methods, omitting helper functions, third-party classes, and standard ROS utilities.

Figure 32: UML class diagram of the main classes and their attributes, functions, relationships, and associations.
Project Management Tools

Slack

Figure 33: A very typical weekly digital standup from mid-February.

Slack’s channels makes it easy to logically divide conversations. Among other things, we recorded all our weekly digital standup meetings from the first half of the project in Slack.
Meeting Minutes

![Meeting Minute template illustration]

Figure 34: Illustration of Meeting Minute template and some of our output.

We used meeting minutes during important meetings to log key decisions, record advice from our supervisor, and ensure aligned understanding of deadlines. The entire log of Meeting Minutes (about 22 pages) is available upon request or at this link: [https://goo.gl/xL7s6J](https://goo.gl/xL7s6J). Over time, we regularly adapted our logging tools depending on the rhythm of our project, leaning particularly on Asana, Slack, and WhatsApp.
Division of Work

<table>
<thead>
<tr>
<th>Components</th>
<th>John Lingi</th>
<th>Sagar Doshi</th>
<th>Kim Rants</th>
<th>Clara Pouletty</th>
<th>Nico Smuts</th>
<th>Fabian Falck</th>
<th>Total Time Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Machine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Medium</td>
</tr>
<tr>
<td>Command Input</td>
<td>10%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Medium</td>
</tr>
<tr>
<td>Audio Out</td>
<td>10%</td>
<td></td>
<td>50%</td>
<td></td>
<td>10%</td>
<td></td>
<td>Low</td>
</tr>
<tr>
<td>Mapping and Navigation</td>
<td>30%</td>
<td>10%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>High</td>
</tr>
<tr>
<td>Object Recognition</td>
<td>30%</td>
<td>50%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>High</td>
</tr>
<tr>
<td>Grasping</td>
<td>40%</td>
<td>40%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>High</td>
</tr>
<tr>
<td>Face Recognition</td>
<td></td>
<td>10%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Medium</td>
</tr>
</tbody>
</table>

| Total                       | 100%       | 100%        | 100%      | 100%           | 100%       | 100%         | Medium                 |

Figure 35: Table showing how hours were divided among team members per component.

The above table displays an overview of hours spent across components. The rightmost column gives a relative estimate of how much time, as a team, we spent on that particular component. It is important to stress that the term “Components,” both here and in Section 6, does not represent an exhaustive list of all the tasks we accomplished. For example, code review, documentation, report writing, parts of testing, stakeholder interfacing, project presentation, and more are not included. However, this does indicate where our team devoted most of our hours. We hope this helps future project teams who need to allocate their resources.
ROS Illustrations

Sample of Robot DE NIRO ROS Node Graph

Figure 36: Illustration of nodes, adjacencies, and connections while running DE NIRO. This graph information is held in the ROS Master, though this image only a sample of the total nodes running.

The ROS Master, as described in Section 5, manages an address lookup table as a node adjacency list. Multifarious nodes keep the Baxter arms running in the background. In our project, we added specific nodes to this graph, sending or receiving messages across this network.
Sample of ROS Topics While Running Robot DE NIRO

```
1 /cameras/left_hand_camera/camera_info
2 /cameras/left_hand_camera/camera_info_std
3 /cameras/left_hand_camera/image
4 /cameras/right_hand_camera/camera_info
5 /robot/end_effector/left_gripper/command
6 /robot/end_effector/left_gripper/configure
7 /robot/end_effector/left_gripper/properties
8 /robot/end_effector/left_gripper/rsdk/set_properties
9 /robot/end_effector/left_gripper/rsdk/set_state
10 /robot/end_effector/left_gripper/set_object_mass
11 /robot/end_effector/left_gripper/state
12 /robot/limb/left/collision_avoidance_state
13 /robot/limb/left/collision_detection_state
14 /robot/limb/left/command_joint_position
15 /robot/limb/left/command_nullspace_setpoint_and_twist_stamped
16 /robot/limb/left/command_stiffness
17 /robot/limb/left/command_stiffness_limit
18 /robot/limb/left/command_twist_speed_limit
```

Many of these topics map to the nodes in the previous image. However, nodes and topics are different objects within ROS. Nodes are connected to one another via the ROS Master. Some of those publish or subscribe to data on particular topics. This is a sampling of the active topics after initialising DE NIRO.